Strong Consistency in a Geo-Replicated File System

Benjamin Bengfort
University of Maryland
bengfort@cs.umd.edu

Pete Keleher
University of Maryland
keleher@cs.umd.edu

Implementation

The FluidFS File System is implemented in Golang as a filesystem in user space [4] using the `bazil.org/fuse` [5] implementation in pure Go (without libfuse bindings). This dependency is a custom implementation of the kernel-userspace communication protocol inspired by the FUSE library.

FluidFS mounts one or more mount points defined in a host-specific, `fstab`-like configuration file. Each mount point constructs an independent file system from the perspective of the kernel, whose top level is the root directory where the FS is mounted. FluidFS, however treats each mount point as a subdirectory of an abstract global file system, specified by a unique *prefix* or TLD (top level directory) that must also be identified in the configuration file. When two mount points are specified with the same prefix (either locally or on different hosts), FluidFS treats each mount point as a partial replica of the global file system.

Interaction with the file system occurs either through `ioctl` calls from the kernel via FUSE or through a RESTful API over HTTP. The primary FluidFS process is therefore composed of several servers: FS servers for FUSE for each mount point, an HTTP server for the REST API, and replica servers for both blob and metadata replication. The API exposes standard create, read, update, and delete CRUD operations for interaction with file objects by implementing standard HTTP methods.

Data Model

FluidFS is a virtual distributed file system that does not manage a disk directly but instead primarily manages objects (files). A file is identified by a unique path and is composed of a collection of version metadata and binary blobs that are stored separately. The filesystem lists the latest version for a file and optionally can display the version history for a specific file. Directories in the file system are computed based on prefixes of paths in the file system and are currently only cached locally. Empty directories therefore are not currently replicated.

There are also two primary data stores utilized by the file system: a version store and a blob store. Versions are stored in a local embedded key/value database for fast metadata reads and writes to disk. Currently interfaces for both LevelDB and BoltDB are implemented for the version store. Blobs are stored on disk (location configurable) in a hierarchical format that provides for fast search and retrieval of blobs.

File Versions

A file is composed by multiple versions and associated binary data. A *file version* is represented as a single piece of meta data that contains at minimum two Lamport scalar compound numbers [2]: the version number and the parent of the current version, as well as an ordered list of blob identities that contain the data for the file. Versions can also contain other metadata such as user information, encryption, access time and statistics, or permissions. A file can read from a single version meta data, though that read may be stale.

When a file is created a new version with no parent and no blobs is created to represent the root of the file. When files are written to, a new version is created for the write whose parent is the version that was previously read or cached locally. The versions that represent the file are therefore a tree of totally ordered writes whose conflict-free version numbers show a total ordering of file operations. Replication and consistency concerns are handled when the newly created version is flushed to disk.

File version information is stored in a local cache implemented by an embedded key/value database. Each database contains three ”buckets” (unique keyspace partitions): `names`, `versions`, and `prefixes`. The `names` bucket stores the current *view* of the database, that is it maps file paths to `Version` objects. The `versions` bucket stores the file version meta data as well as pointers to other versions. The `prefixes` bucket is a local cache that stores information about the contents of directories.

Therefore to read a locally cached copy of the file, you lookup the current version of the file in the `names` bucket, then you fetch the version from the `versions` bucket. The file metadata informs the system if you have permission to read the meta, and if so, the blobs are fetched from disk.

Blobs

A blob is a bounded length array of binary data that can belong to one or more versions of one or more files. Blobs are unique and immutable in the file system and are identified by the hash of their contents. Blobs are detached from file versions such that they are stored and replicated independently, decoupling their distributed behavior. This allows optimism and a separation of concerns: consistency depends only on version replication, durability on blob replication and so forth.

When the a file is flushed to disk, it is chunked into blobs using either fixed length chunks or variable length chunking using a Rabin-Karp rolling hash [1]. Variable length chunking is preferred since it reduces the number of blobs in the system, thereby reducing storage overhead costs. The identity (hash) and order of the blobs that compose the file are stored in the version meta data and the blobs themselves are written to disk.

We currently compute the identity of the blob as a base64 encoded cryptographic hash (digital signature). We specify several options for hashing algorithm including MD5, SHA1, SHA224, SHA256, City Hash, Murmur, and Sip Hash (SHA256 is the default). Each cryptographic hashing algorithm provides trade-offs between performance and likelihood of collisions.

Blobs are stored on disk in a data directory, organized hierarchically by prefixes of their base64 encoded hash value. Currently we prefix blobs by 7 characters constraining the depth of the tree to 3 levels. The root of the data dir therefore contains folders with the first seven characters of the blob's hash, each of which contain the next 7 characters and so forth. The location of a blob on disk is deterministically computed based on its hash and the configuration of the local host. This data structure allows us to create a pseudo-Merkel Tree [3] for efficient detection of changes in the underlying blob store to support anti-entropy replication of blobs.

Consistency

File versions represent the state of the file system and must be replicated in a strongly consistent fashion to avoid two primary forms of inconsistency: stale reads and forks. A stale read occurs when a read access occurs at the local cache even though a more recent update exists elsewhere in the distributed system. A fork occurs when two concurrent write accesses create two independent versions with the same parent. Both stale reads and forks are symptomatic of weak or relaxed consistency mechanisms that allow concurrent access to objects in the distributed system and resolve conflicts as they occur.

Sequential consistency can be observed globally on a per-object basis because there will be a single, linear version history. Inter-object consistency requires file system-transactions where a series of updates includes dependency information. These dependencies are stored in the version object and a read request cannot be satisfied unless dependencies have been satisfied. Inter-object consistency can therefore be observed by directed, acyclic graphs of dependency structures.

Whether or not transactions are implemented for writes, a consensus algorithm must govern the accesses to the file system to ensure that strong consistency is maintained. Consensus has two phases: propose and commit in both phases all participants in the quorum must ensure that file system invariants are met, for example not allowing forks to occur. For strong consistency we propose two governing mechanisms: lease (lock) commits and access commits.

Lease Commits

The first mechanism is perhaps the simplest on the surface – replicas in the file system implement a distributed lock server via consensus. When a host would like to access a file or set of files, it requests a *lease*, a timed lock, from the leader of the consensus protocol. It can then perform any operations on the files specified by the lease. Before the lease expires, the host must commit its final writes and release the lock. If the lease expires, the system is reset to the original state and all uncommitted writes during the lease period are dropped.

When the client requests the least, it must demonstrate that it is as up to date as the leader, otherwise the lease request will be rejected (or the client will be made up to date with the latest state of the file system). The client can request a single object in the file system by path or a portion of the file system using glob-like syntax e.g. `/docs/proj/*`. A consensus decision is made and if the lock request is committed then the client may access the files under its lease.

During the period of the lease, any requests for files under the lock will either be blocked by the leader or rejected. The consequence of contention in this system is latency or no progress. To ensure the system is not deadlocked, if the leader has not received a heartbeat message from the client holding the lock and another lock request has become available, then the leader will forfeit the current lock and no writes made under that lock will be accepted. If no requests for the lock are made, it is possible for the client to recover and commit changes after the lock has expired.

Once the client is done with its accesses, it submits a lock release request. The lock release request contains any updates made to the underlying file system. The lock release and updates are committed, and once committed the lock is released. Clients can also request a lease extension, which also includes a snapshot of current writes to ensure that the system remains durable.

Commands that are applied to the state machine:

1. `ACQUIRE LEASE`: requires client to be as up to date as the leader for the files requested in the lease.

2. `FORFEIT LEASE`: called by the leader to cancel a currently held lease.

3. `WRITE VERSIONS`: during a lease, files accessed during the lease can be committed. This must happen before the lock is released.

4. `RELEASE LEASE`: this must be committed to ensure that all replicas know the lease is released and can be handed out again.

5. `EXTEND LEASE`: essentially RELEASE/ACQUIRE in one step, this is only allowed if no other client has requested a lease.

Leases can be extended to implement read and write leases and other mutex semantics.

Access Commits

Leases are ideal in the scenario where one client needs to make many changes during a single session, however, leases maintain strong consistency by blocking concurrent access for a fixed window of time. A more optimistic approach is *access commits* where consensus decisions are made on actual file system operations. In this case the file system implements *close to open* consistency rather than per-access consistency because low granularity will require too many consensus operations. The access commits mechanism is optimistic in that it believes that conflict is rare and that when conflicts do occur they can be easily resolved.

When a client wants to modify an object in the file system it makes a request to the leader that initializes a consensus decision to serve the access request. If a client wants to write a new version to a file, it submits the write; if the write is not a fork – that is its parent is the latest committed write, the leader accepts the write, otherwise the write is rejected or must be prepared. Forks occur if on open, the version that was read becomes stale while the file is being edited e.g. another write was committed. This is more likely if a local cached version is read to open the file, it is less likely if a `READ` access was specified to the leader, who returns the last committed value.

We can therefore specify how we want reads to be handled to decrease the likelihood of contention at the cost of read latency. Read local cache is the fastest but most likely to have a conflict. A remote read to the leader of the consensus group that returns the latest commit or the latest proposed write is even less likely to have a conflict, and finally a remote read that is committed by the consensus protocol will result in external linearizability of all reads and writes to the system.

If a write is a fork, the leader has several options; if we simply reject the forked write then we have implemented a ”first writer wins” policy and the application will have to handle the conflict. For text files, it is possible to use a merge protocol similar to Git to repair conflicts, or for images and videos which are usually updated atomically the system can simply choose to overwrite the file with the new data, updating the parent as required.

Commands applied to the state machine are as follows:

1. `CREATE`: create a file, if it exists already the operation is rejected.

2. `WRITE`: commit a new version of the file, it is rejected if the version is a fork. Writing to a file that is not created will be rejected unless a flag is set, then two commands are issued together.

3. `DELETE`: delete the file from the file system, a write whose parent is deleted is rejected unless a flag is set in which case the file is created again with a new root version.

4. `READ`: optional, read accesses do not need to be committed unless an external linearizability is required.

While leases are simpler to implement with respect to transactions, access commits can also maintain transaction semantics by writing versions with dependency information included.

Replication

Because file versions specify the *view* of the file system, only file version meta data needs to be replicated consistently for correctness. The underlying data, residing in blobs, can be replicated orthogonally to the version information and requires no correctness checks. By separating the replication and consistency of file versions and associated data, we hope to show improvements to the performance of the system as well as strong guarantees for both consistency and durability.

Version Replication

We propose to use both the Raft consensus algorithm as well as an extension of Raft, Alia (hierarchical consensus) to replicate versions with strong consistency. In this case, versions are replicated by being broadcasted via the consensus protocol, note that in this case all replicas must participate in consensus. Changes are applied to the view of the file system on commit, new replicas can be brought up to date with the state of the leader by performing a snapshot repair of their log, then replaying all log commands.

For comparison we propose the following topologies:

1. *Raft with writes*: Raft consensus with optimistic access commits.
2. *Raft with leases*: Raft consensus implementing a distributed lease server.
3. *Alia with both*: Hierarchical consensus where the root quorum partitions the namespace (essentially high level leases) to subquorums that can be *either* write or lease quorums depending on the requirements for the partition and the number of clients.

Note that because file versions are small pieces of metadata, file versions are *totally replicated* across all replicas.

Blob Replication

The chunks that compose a file (blobs) are immutable and therefore can be safely replicated using *anti-entropy*. This mechanism ensures that large amounts of data are not slowing down the consistency process, that blobs can be partially replicated to save space, while still ensuring a high level of durability. When a host tries to read a file, if the blobs haven't been replicated, they can demand-fetch the blobs from the originating client or a *sync* node – a host with a large disk that is prioritized in anti-entropy. Additionally we can use heuristic or learning algorithms to colocate blobs where they are most likely to be read.

We propose *bilateral anti-entropy* such that every anti-entropy session repairs the state between both the local and remote. On a routine interval, the localhost will select a remote peer to synchronize with (*sync nodes* will have a slightly higher probability of being selected). The local machine will send a tree of the prefixes of the hashes of the blobs (the same tree that blobs are stored on disk), each node in the tree will have the count of the number of blobs underneath it. Using this data structure, it is is simple to identify which directories have been updated with new blobs, and the remote returns a listing of those directories. The local will then send all missing blobs along with a request for blobs it is missing.

Note that by simply using counts it is possible that two different blobs will be inserted into the same directory, causing both replicas to assume that they are up to date. The first mechanism to handle this is to ignore counts in the root of the tree - comparing only counts and the second level. In the rare case that this does occur, on read the blobs can be demand fetched. If stronger guarantees are required, each level of the tree can also track the hash of the hashes of all its children, which also makes it easy to detect changes.

Network Environment

FluidFS is deployed on a geographically distributed cluster of commodity hardware in two countries and two continents. Currently, we are running FluidFS on six dual core machines with a minimum of 8GB of RAM and 256GB of hard disk space. Four of the machines are located in College Park, Maryland one is located in Seattle, Washington and the last machine is located in Athens, Greece. Each machine runs a single FluidFS server process with multiple mount points.

Each machine is connected to a broadband internet connection with a minimum of 100 MBit/second download and 5 MBibt/second upload guaranteed bandwidth. The network is provided by standard residential ISPs with no dedicated access. The network is prone to increased latency especially during peak demand times; it is also prone to partitions – short periods of network unavailability. These conditions can be further amplified programmatically to demonstrate the behavior of our system under adverse conditions.

API

This section discusses the HTTP API for interacting with the FluidFS server; the other client API is the FUSE API, which provides for standard FS access using kernel `ioctl` commands inside of a mounted directory. Note also that the HTTP API is not currently fully implemented.

FluidFS runs an HTTP server that maps standard HTTP requests to paths in the file system. For example, the HTTP request `GET /bbengfort/docs/foo.txt` returns the file from the `/bbengfort` prefix at `/docs/foo.txt`. A request to the API contains an HTTP verb which defines the operation, HTTP headers modify the operation, the path of the request defines a location in the file system, parameters in the URL modify the request and finally the HTTP body contains data.

A response to a request contains an HTTP status code (e.g. `200: OK` for success), headers that modify the response, and a body that contains the requested data.

The primary HTTP verb is `GET` which specifies a read access to the file system. If the path of the request ends in a `/` the server interprets this as an listing request for a directory, which simply displays an HTML page with the contents of the directory, otherwise it assumes the request is for a file object. If the file doesn't exist the response will be `404: Not Found`, otherwise the type of response will depend on the value of the `Accept` header in the request which specifies a mimetype for the response as follows:

1. `Accept: text/html` a web page is returned that displays the current file meta data in a human readable fashion.

2. `Accept: application/json` or `Accept: application/protobuf` the server will return the version data serialized in the specified machine-readable format.

3. `Accept: application/octet-stream` will cause the server to compose the file from blobs and return the file as an attachment for download.

Other HTTP request headers can include authentication/authorization information, partials and ranges for multipart downloads, requests for encoding, etc.

HTTP parameters also allow deeper control of the request. For example, the `?version=9.3` parameter allows the user to fetch historical versions of the file. More importantly the `?blob=a3de93af` parameter allows the user to fetch a specific blob and is useful for demand-fetching blobs before they are replicated locally. The `?blob` parameter can also be used at the root to request a blob not associated with a specific file, though in this case no file-specific validation occurs nor is there a guarantee of success.

Other HTTP verbs specify various *accesses* on the server as follows:

1. `POST`: create a file at the specified path. If the file does not exist and the request includes data, this is the same as create and update in one step. If the file exists and data is included, the server will return a `409: Conflict` response. If no data is included in the request, then this is equivalent to `touch`.

2. `GET`: read a file or list a directory.

3. `PUT`: update (write) to a file at the specified path. A successful request will return the newly created version similar to how a `GET` works. This method expects the entire file to be uploaded.

4. `DELETE`: delete the file at the specified path.

5. `HEAD`: collect information about what will be returned from a read request, e.g. the size of the data or status of the response. Basically a `GET` without a body.

The HTTP API is fairly straightforward using well known resource operations, standards, and codes. Hopefully this will ensure that the API is easy to use and create clients for, whether on the web or on the command line.

Configuration API

I propose a special path for the HTTP API, `http://host:port/etc`, that is used to configure FluidFS both locally for the specified host or globally for the entire file system. I considered using subdomains, but FluidFS has no control over routing at the domain level, or using a different port which would involve instantiating another HTTP server alongside the API server; in both cases it felt simpler to simply reserve `/etc` as a special prefix in the file system.

This path could provide a web interface for managing the server as well as viewing status and statistics information. Alternatively we could write configuration and status files in this directory, managing them similarly to other files in the file system with the exception that only the system was allowed to write to this directory.

References

[1] Karp, R. M., and Rabin, M. O. Efficient randomized pattern-matching algorithms. 249–260.

[2] Lamport, L. Time, clocks, and the ordering of events in a distributed system. 558–565.

[3] Merkle, R. C. A digital signature based on a conventional encryption function. In *Conference on the Theory and Application of Cryptographic Techniques*, Springer, pp. 369–378.

[4] Vangoor, B. K. R., Tarasov, V., and Zadok, E. To FUSE or not to FUSE: Performance of user-space file systems. In *FAST*, pp. 59–72.

[5] Virtanen, T. bazil.org/fuse.